The Thesis Committee for Rodolfo Kaplan Depena
Certifies that this is the approved version of the following thesis (or report):

Diamond: A Rete-Match Linked Data SPARQL Environment

APPROVED BY
SUPERVISING COMMITTEE:

Supervisor:

Daniel Miranker

Don Batory

Diamond: A Rete-Match Linked Data SPARQL Environment

by

Rodolfo Kaplan Depena, B.A., B.S.C.S.

Thesis

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Computer Sciences

The University of Texas at Austin

December 2010

Dedication

To my mother Martha Kaplan, my father Rudolph Depena, my sister Christina Depena, my girlfriend Christine Garwood, my best friend Oren Freiberg, and the Moore-Hill Residence Hall students and staff (especially Tanya Lowery). I am honored and humbled to have such wonderful friends and family.

Acknowledgements

I wish to thank the faculty and staff at UT-Austin for the opportunity to be educated at a wonderful institution. I wish to acknowledge and thank Daniel Miranker, who allowed me to grow as a scientist and helped me put this thesis document in a coherent fashion. He also taught me the meaning of brevity. I wish to thank Don Batory, who kindly agreed to read this thesis and introduced me to the wonderful world of databases as an undergraduate. Kathryn McKinley also deserves my thanks and praise. She encouraged me and offered me the opportunity to see what successful research can look like in her research group. Additionally, I never would have been able to construct a suitable compiler for Diamond without the foundational knowledge Kathryn taught me during our advanced compilers and memory management courses. I would like to thank my first research advisor, Greg Lavender, who has been an academic father of sorts. He has given me advice when I've been lost in the complexity of our industry, mentored me through an undergraduate honors thesis, has been a consistent supporter of my research/academic pursuits during my time as an undergraduate and graduate student, and has given me a thirst for research that will not be quenched. You can't find a more diligent scientist or better mentor. I thank Hyunjoon Jung for allowing the use of his graphical SPARQL debugger that allowed me to debug my software. Also, I thank Oren Freiberg for showing me the openly available Java Tree Builder, which I use to develop syntax trees in my compiler. I thank Juan Sequeda for our brief discussion on SPARQL and for using my software.

3 December 2010

Abstract

Diamond: A Rete-Match Linked Data SPARQL Environment

Rodolfo Kaplan Depena, M.S.C.S

The University of Texas at Austin, 2010

Supervisor: Daniel Miranker

Diamond is a SPARQL query engine for linked data. Linked data is a sub-topic of the Semantic Web where data is represented as a labeled directed graph using the Resource Description Framework (RDF), a conceptual data model for web resources, to affect a web-wide interconnected, distributed labeled graph. SPARQL graph patterns entail portions of this distributed graph. Diamond compiles SPARQL queries into a physical query plan based on a set of newly defined operators that implement a new variant of the Rete match, a well known artificial intelligence (AI) algorithm used for complex pattern-matching problems.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Linked Data is an aspect of the Semantic Web intended to treat data as a distributed interconnected graph [7, 10] just as the web, through hyperlinks, has enabled documents to be interconnected and distributed. In the linked data model, individual graph edges are defined in Resource Description Framework (RDF) and are known as triples [25]. A triple is a single graph edge, comprised of a subject, a predicate and an object, each represented by a URI. Objects may also be literals. A triple stored on one computer may comprise a URI, *u*, whose domain is another computer. Figure 1(a) contains a set of triples in RDF. Figure 1(b) illustrates the same triples as a graph [25] that spans multiple computers.

Much like a user may click on a URL embedded in a document and a new document from another web site may appear in their browser, in Linked Data, *u* may be dereferenced and the computer identified by *u* will return one or more additional triples. Linked data queries are evaluated by crawling, filtering and dereferencing in a repeated cycle until a fixed point is reached [21].

Linked data queries can be expressed in the SPARQL graph query language [31]. Figure 2(a) illustrates a SPARQL query that when evaluated will return a set of triples, (see Figure 2(b)) that comprise a sub-graph of the graph presented in Figure 2(b).

Diamond is an engine for SPARQL query evaluation over linked data. URIs embedded within the SPARQL query comprise the starting points for crawling the graph of linked data. The first step in a Diamond evaluation of a SPARQL query is to

dereference those URIs. In the case of Figure 2(a), URIs from the doefamily, dbpedia, and friend of a friend are dereferenced.

```
#Graph from http://www.doefamily.com/
@prefix doefamily: <http://www.doefamily.com/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix dbpedia: <http://dbpedia.org/resource/> .

doe:John rdf:type foaf:Person .
doe:John owl:sameAs dbpedia:John_Doe .
doe:John foaf:age 72 .
doe:John foaf:mbox <mailto:john@doe.com> .

doe:Jane rdf:type foaf:Person .
doe:Jane owl:sameAs dbpedia:Jane_Doe .
doe:Jane foaf:age 12 .
doe:Jane foaf:phone “512-475-6656” .
```

Figure 1a: Example of RDF Triples written in Notation 3

For the purposes of our discussion, *dereferencing* is the process of extracting triples from a triple store housed on a server and identified by a URI. Diamond compiles a SPARQL query into a Rete network, which processes triples in a manner understood to be equivalent to both query processing and complex pattern-matching. The triples that fully satisfy the graph patterns and filters within the query, as expressed by the Rete network, form a resulting set of solutions; the answer to the query. As URIs appear in triples processed by the system, URIs add to a queue in FIFO order to be dereferenced later. The system is designed such that in future work, URIs may be dereferenced by applying priority schemes that integrate concepts such as trust and page rank. The triples that fully satisfy the SPARQL query are depicted in Figure 2(b).

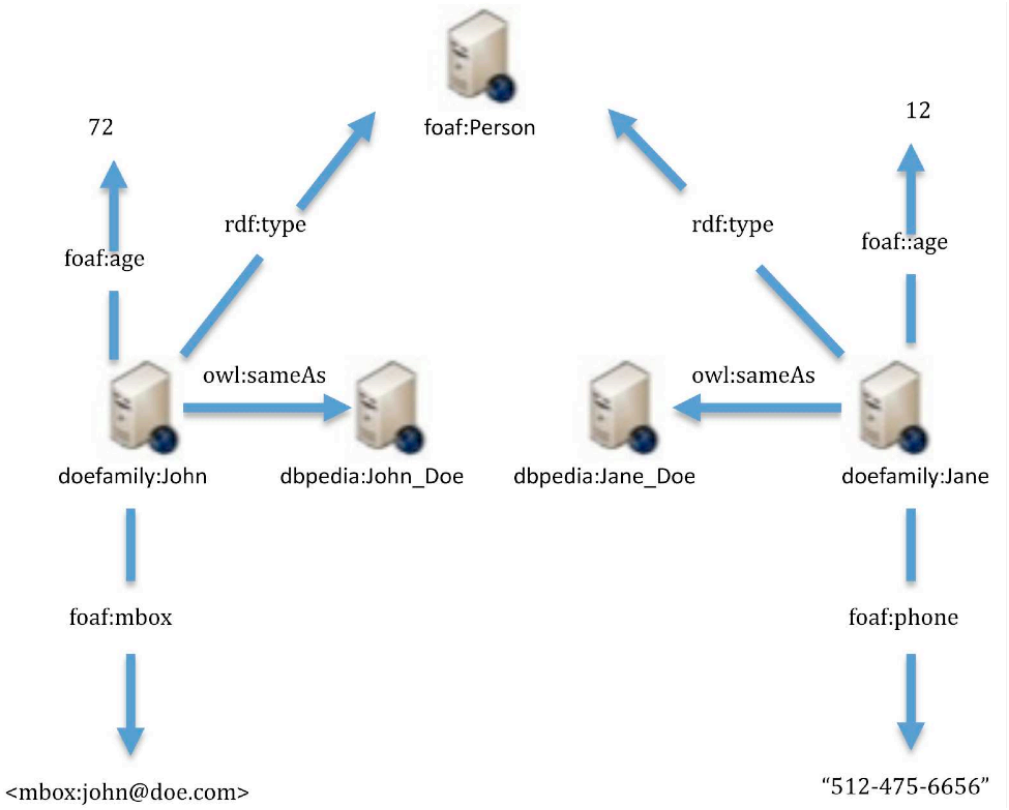

Figure 1b: Labeled directed graph of RDF data in Figure 1a

We observe a similarity in the execution model of linked data queries and forward-chaining rule systems in AI. In the latter, a set of rule antecedents is evaluated against a repository of *working memory*. In the case of linked data, working memory refers to a set of RDF triples. A satisfied rule is selected and its consequent executed. The consequent may insert additional working memory elements. In SPARQL, triple patterns are considered to be akin to production rule antecedents with no right hand side consequent. Production rule sets antecedents are reevaluated in an indefinite cycle. Thus, we liken dereferencing a URI and the additional triples fetched, to the firing of a rule that adds elements to working memory. In the analogy, the rule-based program contains a

single rule; a rule whose antecedent corresponds to the SPARQL query and whose consequent dereferences the URIs that appear in any triples satisfying the query.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/> .
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
PREFIX dbpedia: <http://dbpedia.org/resource/> .
PREFIX owl: <http://www.w3.org/2002/07/owl#> .

SELECT ?x
FROM NAMED <http://www.doefamily.com/>
WHERE{
 {
  ?x rdf:type foaf:Person .
  ?x owl:samAs dbpedia:John_Doe .
  ?x foaf:age ?age .
  FILTER(?age > 64) .
 }
 UNION
 {
   ?x rdf:type foaf:Person .
   ?x owl:sameAs dbpedia:Jane_Doe .
   ?x foaf:phone ?phone .
   OPTIONAL{?x foaf:mbox ?mbox .}
   FILTER(!bound(?mbox)) .
 }
}
```

Figure 2a: SPARQL query that finds people old in age (those over the age of 64) and old at heart (those who have no email addresses)

Forgy's Rete match is the de-facto standard for implementing forward-chaining rule systems [16, 17]. Rather than re-evaluating the rule antecedents at each cycle, the Rete match processes incremental changes to the working memory as incremental changes to the set of satisfied rules. This is accomplished by interposing state operators, or *memory nodes*, in between a network of filtering operators. Incremental changes to working memory are processed as cascading incremental changes to the memory nodes.

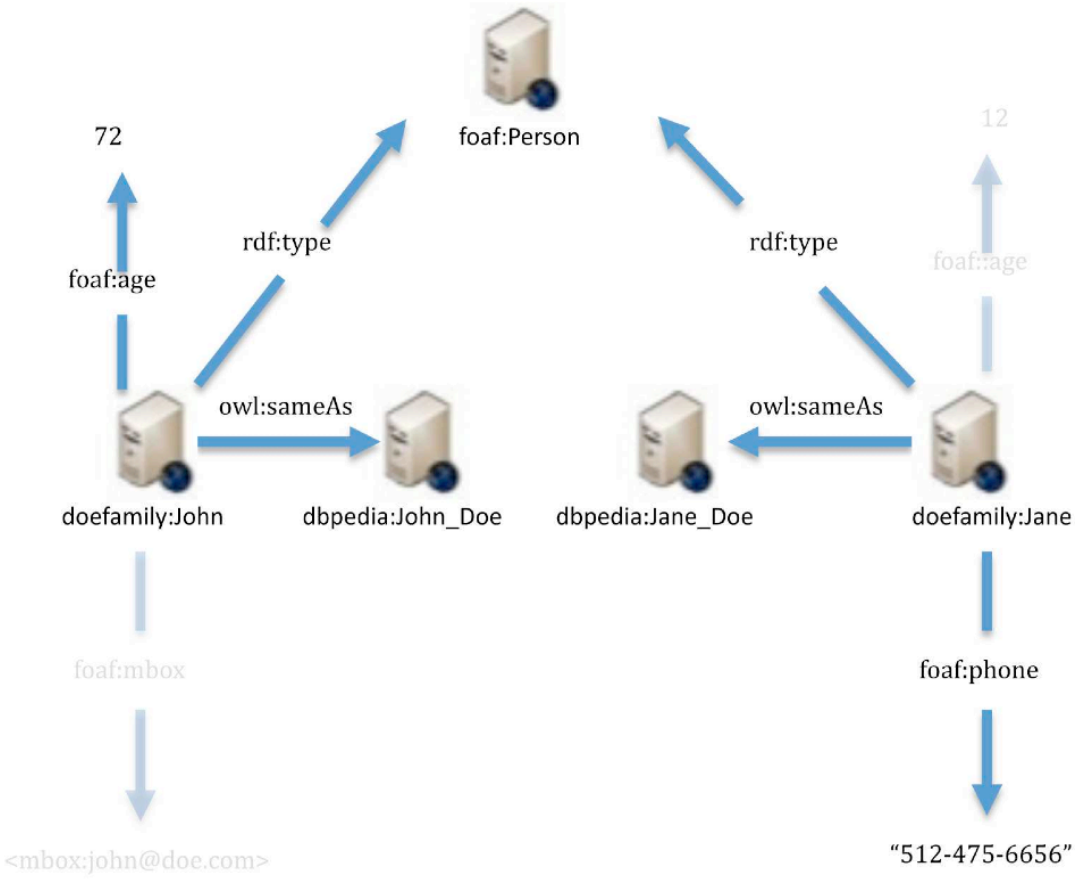

Figure 2b: A linked data evaluation of the SPARQL query in Figure 2a

The primary contribution of this paper is the definition of a set of Rete operators capable of executing SPARQL queries. When defining a new rule language it is common to define a new Rete operator set. We note that the OPS2 rule language represented working memory as arbitrary LISP expressions. In the OPS5 language, working memory elements have a decidedly relational table like structure. Both languages were implemented by Forgy using the Rete match [16, 17]. Miranker leveraged the similarity of OPS5 working memory to relational tables, to cast rule evaluation in the form of relational query evaluation and apply relational database optimization methods to the execution of rule systems [29]. The similarity of the execution model supported by the

Rete match and the requirements of stream databases has led to at least one stream database system that uses the Rete match [24]. Also, the semi-naive evaluation of Datalog programs is another example of research that draws a parallel between rule evaluation and query evaluation; one that we will exploit due to the results of [2].

Chapter 2

The Rete-Match Algorithm

As details of the Rete match are well represented in the literature [16, 17] we provide only a short introduction to the algorithm here, by means of an example. We consider an example of an OPS-like query as shown in Figure 3(a). The same OPS rule written in Figure 3(a) can also be represented as a SPARQL query in Figure 3(b). In turn an equivalent query on self-joined views is expressed in SQL as seen in Figure 3c. Figure 4(a) represents the state of the network before a right hand side token enters the network, whilst Figure 4(b) represents the state of the network after the right hand side token enters the network.

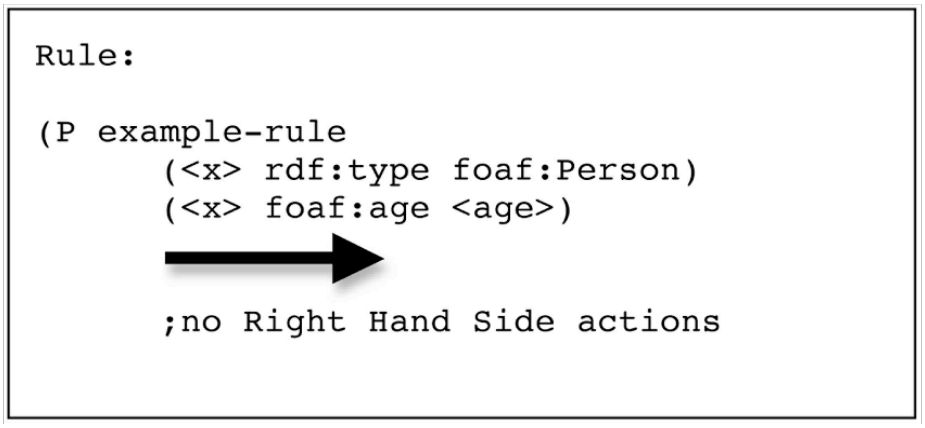

Figure 3a: Example OPS-like rule[1]

The Rete match compiles rule antecedents into a network that can be characterized as a physical query plan where the query operators are separated by

[1] Prefix notation is not common to OPS rules, however we have kept the prefixes for the sake of continuity and the benefit of a semantic web audience.

memory nodes that represent the materialization of the sub-query antecedent to the memory node. Figure 4(a) illustrates the state of the Rete network after three triples have been added. On the left the vertices are labeled using the Rete terminology associated with the OPS5 implementations, on the right with the operator names used by Diamond. Figure 4(b) illustrates the changes and flow when adding a fourth triple on the right hand side.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/> .
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

SELECT *
WHERE {
   ?x rdf:type foaf:Person .
   ?x foaf:age ?age .
}
```

Figure 3b: Example SPARQL Query translated from Figure 3a

```
SELECT t1.s as x, t2.o as age
FROM tripleview t1, t2
WHERE
t1.p = 'rdf:type' AND
t1.o = 'foaf:Person' AND
t1.s = t2.s AND
t2.p = 'foaf:age'
```

Figure 3c: Example SQL Query translated from Figure 3b

In Rete match parlance, the data that flows through the network are dubbed tokens. In Figure 4(b), a token representing the triple that defines Jane Does age enters the network. It is compared to the `<x> rdf:type foaf:Person` and `<x> foaf:age <age>` test nodes. It does not satisfy the predicate in the left select node, but does satisfy the right select node. As a consequence, the token, `doe:Jane`

`foaf:age 12`, passes through the `<x> foaf:age <age>` select node. The token reaches the α-memory node; A copy is stored in the memory node and the token flows to the two-input join node. In an incremental fashion, the token is joined with the contents of the α-memory node, because the variable bindings for the token are consistent. In the example, the resulting join node, (`doe:Jane rdf:type foaf:Person` , `doe:Jane foaf:age 12`) to a the β-memory where a copy is stored and the token continues to propagate.

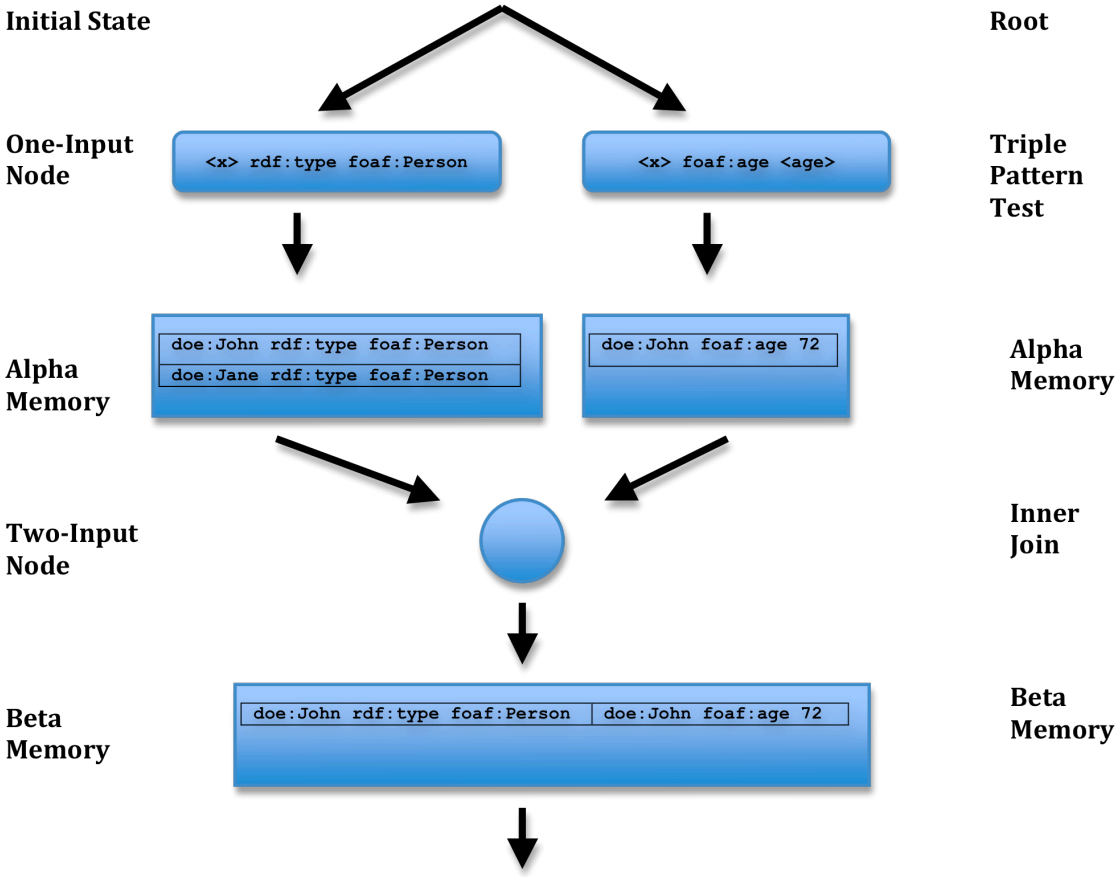

Figure 4a: State of Rete network before Right Hand Side token enters the network. Labels on the left correspond to the terminology used in the original Rete match algorithm. Labels on the right correspond to the names we've chosen to convey the semantics of SPARQL.

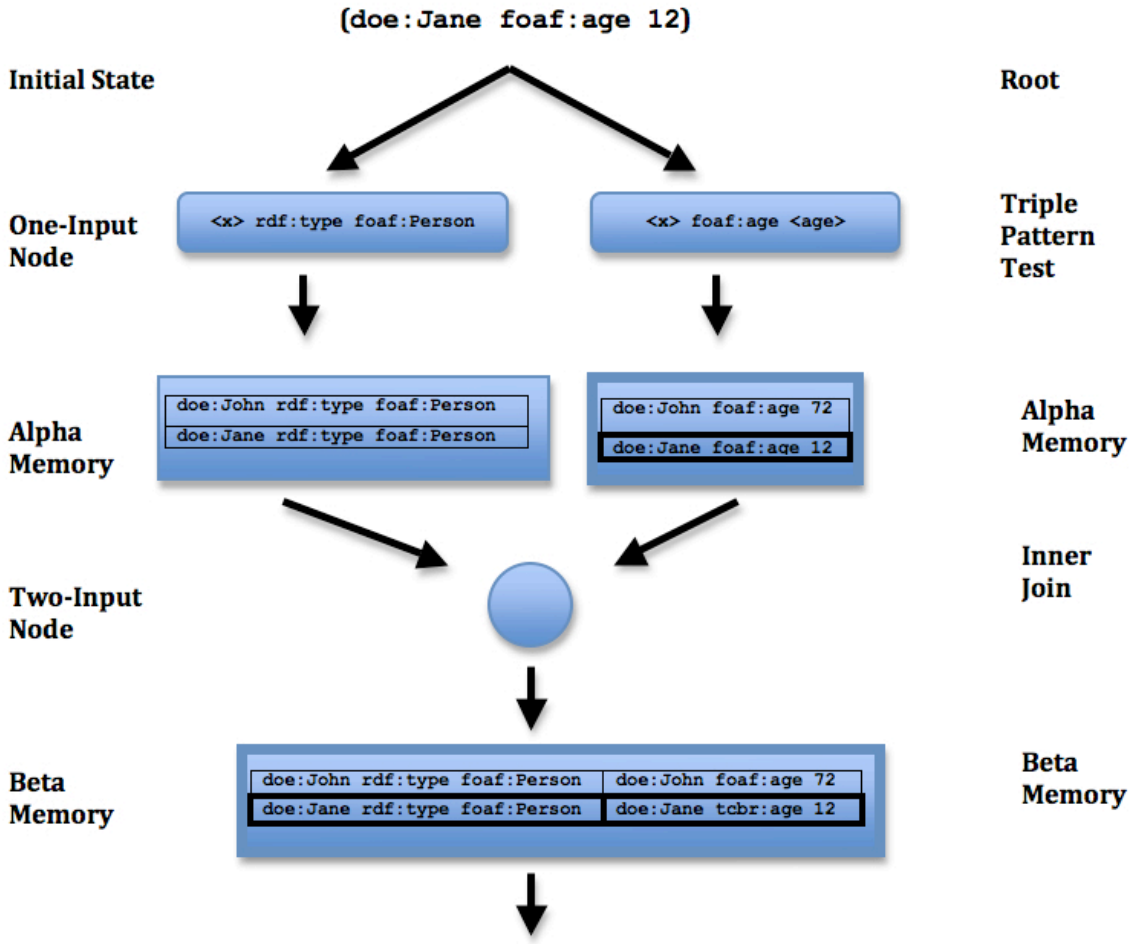

Figure 4b: State of Rete network after Right Hand Side token enters network. Labels on the left correspond to the terminology used in the original Rete match algorithm. Labels on the right correspond to the names we've chosen to convey the semantics of SPARQL.

The reader should note, the final memory node contains pairs of triples that satisfy the OPS-like query as well as the SPARQL query. Further, we point out that, fundamentally, the computation is the same as maintaining a materialized view of the join between the two α-memories, given an incremental change to the database. In relational algebra this is commonly represented as, given two relations, R, S, and a materialized view, $V = R \bowtie S$. If a new record, t, is added to the relation R, such that, $R = R \cup t$,

then the new version of the view, V, may be incrementally determined from V, by computing $V = V \cup (t \bowtie S)$.

Qian and Weiderhold detail the algorithms for correctly, incrementally computing the results of a suite of relational operators [32]. The behavior of incremental recomputation using the outer join operator can be found in [19]. Miranker was the first to relate the operators in the Rete match to incremental evaluation of relational database operators [29]. The equivalence between incremental evaluation of relational operators and rule-based processing is also well explored in the context of the semi-naive evaluation of Datalog.

Chapter 3

Related Work

Our approach is rooted in the utilization of the Rete pattern-matching algorithm. This is not the first time the Rete match has been used in a semantic web context. As might be expected, the Rete match has been used in inference engines such as OWLJessKB[2] and DamlJessKB [26]; tools used to evaluate ontological assertions.

AI and the semantic web are not the only areas where the Rete match has been of use. There is previous work per the use of the Rete match in the implementation of a stream database system [24]. Collections of database triggers such as TriggerMan [20] and Ariel [36] also use the Rete match's pattern matching capabilities to continuously query data through data streams and notify of anomalies through triggers. Because the Rete match is used in multiple areas of computer science, it is important to highlight efforts to improve upon the Rete match through optimization efforts like TREAT [29], parallel Rete match implementations like [3, 27, 35], and lazy match algorithms such as [6, 28].

While Diamond uses the Rete match as a mechanism to evaluate linked data as a data stream, there are other efforts to treat linked data as a data stream [4, 11] and a continuous query effort, which uses an extended version of the SPARQL language [5]. Fortunately, our approach does not require us to extend the SPARQL language to cater to the need of querying linked data. Querying linked data using SPARQL is not a new concept [33] as there are tools such as the Semantic Web Client Library [21] however our

[2] http://edge.cs.drexel.edu/assemblies/software/owljesskb/

approach using the Rete match to allow us to execute SPARQL queries over the web of linked data is new.

There are certain optimizations a developer can take when querying the web of linked data using SPARQL such as query re-writing [12] or caching [23]. Additionally several SPARQL optimization studies have taken place using query models [22] and SPARQL operator complexity analyses [34]. Fortunately, due to the nature of the Rete match, state is incrementally saved during processing iterations and this saved state is liken to a cache that is consistently updated when new tokens are introduced and according to [29] we may treat the Rete network as a physical query plan to be optimized in a traditional relational database fashion, much like optimizing query models [22]. While our work depends on its deep connection to relational algebra, other works attempt to define SPARQL within a relational algebraic context as well [13], but not using incremental recomputation algorithms like we do.

Finally, while our approach is tested through the implementation of a linked data query engine using the Rete match, it is to note that there are other linked data tools available including, but not limited to, the previously mentioned Semantic Web Client Library [21], browsers such as Tabulator [8] and data visualization tools like Explorator [14, 15].

Chapter 4

A Rete Operator Set For SPARQL

SPARQL Language Construct	SPARQL Algebra Operators	SPARQL Operators in [2]	Rete Operators
BGP	eval(D(G), BGP)	T(GroupGP)	TriplePatternTest(tp, R)
(.) a point	JOIN (Ω_1, Ω_2)	P_1 AND P_2	R_1 InnerJoin R_2
OPTIONAL	LEFTJOIN(Ω_1, Ω_2, C)	P_1 OPT P_2	R_1 LeftJoin R_2
UNION	UNION(Ω_1, Ω_2)	P_1 UNION P_2	R_1 Union R_2
FILTER	FILTER(C, Ω)	P_1 FILTER C	Filter(C, R)
SELECT	Project(Ψ, PV)	SELECT	SolutionSequence(s, R)

Table 1: Translation table[3]

Angles and Gutierrez prove that SPARQL is equivalent in expressive power to non-recursive Datalog with negation [2]. Given the relationship between SPARQL, Datalog and relational algebra, to form a Rete operator set for SPARQL one need only, by transitivity, identify a mapping from SPARQL syntax and its native logic, to relational operators. Since Rete operators are physical operators, each such Rete operator must implement an incremental algorithm for its corresponding relational semantics. The translation table in Table 1, details the preceding construction including the relationship between SPARQL constructs, SPARQL algebra, formal semantics, and Rete operators.

[3] Translation table where BGP is a basic graph pattern containing one triple where BGP is a basic graph pattern containing one triple pattern *P, P_1,* and P_2 are graph patterns, *T* is a transformation, GroupGP is a group graph pattern (in our case with exactly one triple pattern). Ω is a multiset of solution mappings, Ψ is a sequence of solution mappings and PV is a set of variables. *C* is an expression. *R*, R_1, and R_2 are relations containing tokens, s is a list of variables.

4.1 THE OPERATORS

For the implementation of the Rete operators we adopt the algorithms detailed by Qian and Wiederhold [32]. Additionally, we use a similar algorithm detailed by Gupta and Mumick [19] for our left outer join operator, *LeftJoin*, to perform incremental updates to the network state. Diamonds LeftJoin operator is consistent with the standard SPARQL specification [31] corresponding to the implementation of the SPARQL OPTIONAL construct.

It is worth mentioning that both our *TriplePatternTest* operator and *Filter* operator are based on Qian and Wiederhold's relational select operator. In the case of the TriplePatternTest operator, this operator selects RDF triples that only match a predetermined triple pattern. This triple pattern is found within a Basic Graph Pattern (BGP) in a SPARQL query where basic graph patterns are sets of triple patterns. The Filter operator is an incrementally evaluated relational select operator that selects RDF triples based on a SPARQL constraint. A formal description can be found in the appendix for both of these operators. *InnerJoin* is also based on the $\bowtie$ operator in Qian and Wiederhold and is used to determine variable consistency before joining two tokens together. If there are no common bound variables between two triple patterns, then a cartesian product is taken.

4.2 NEGATION IN SPARQL AND AN INCREMENTAL ANTIJOIN

Although it has been shown that SPARQL is equivalent to non-recursive Datalog, SPARQL 1.0 does not have an explicit negation operator. Instead, the expression of negation is achieved through an idiom comprising three syntactic features [2], which we detail below. A consequence is, although the original Rete match algorithm contains a NOT operator [16] that implements antijoin, (which in turn is a preferred definition of

negation in Datalog), this operator is not needed for our implementation. An explicit negation operator is anticipated as part of the SPARQL 1.1 specification, and also could be of use as an optimization where the syntactic idiom for negation is identified.

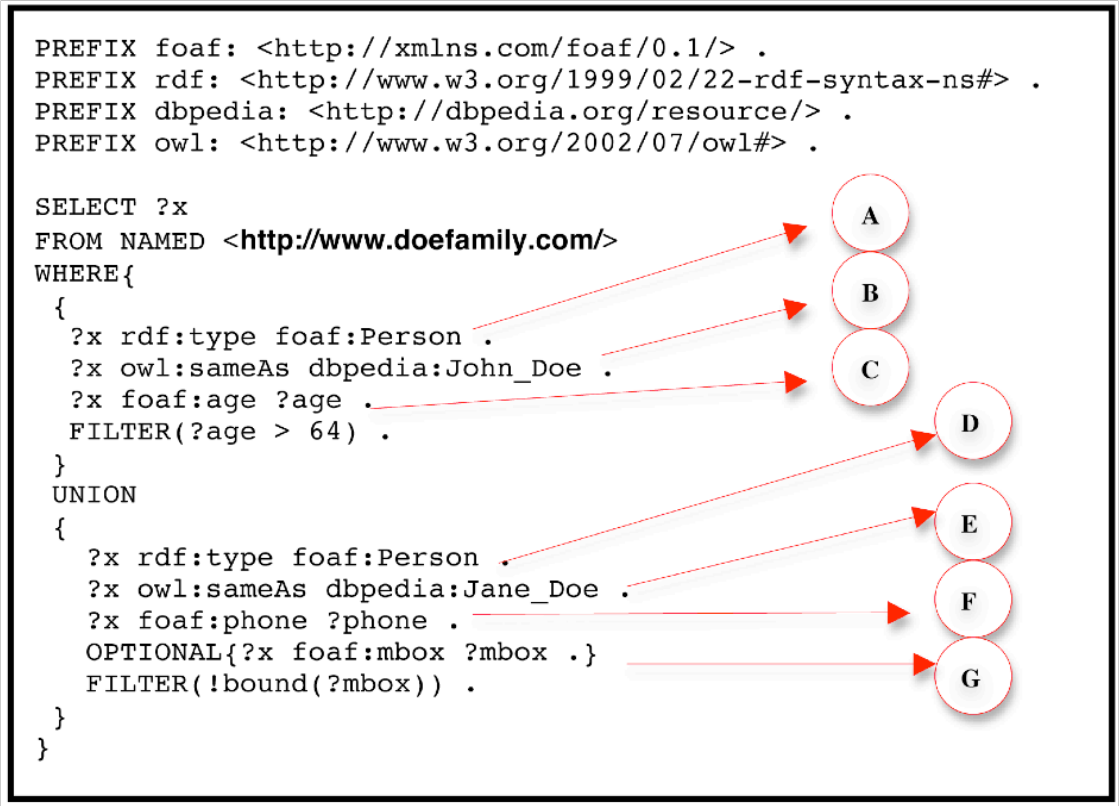

Figure 5a: Annotated SPARQL Query

The SPARQL idioms that implement negation are P_1 OPTIONAL P_2 FILTER !bound(?x), where P_1, P_2, are graph patterns, and ?x is a variable found in an OPTIONAL graph pattern. In the case of the query shown in Figure 5(a), triple patterns *D*, *E*, and *F* are captured in a Basic Graph Pattern representing P_1 in our example pattern. Triple pattern *G* takes the place of P_2 in our example pattern. The filter expression in Figure 5(b), Filter(!bound(?mbox)), also takes the place of !bound(?x) in our example pattern. In our Rete match network example in Figure 5(b), negation manifests itself as β_{DEF}

$LeftJoin_{\text{DEFG}}$ α_G whose result is captured in β_{DEFG} and Filter(!bound(?mbox), β_{DEFG}) whose result is captured in $\beta_{!bound(?mbox)}$.

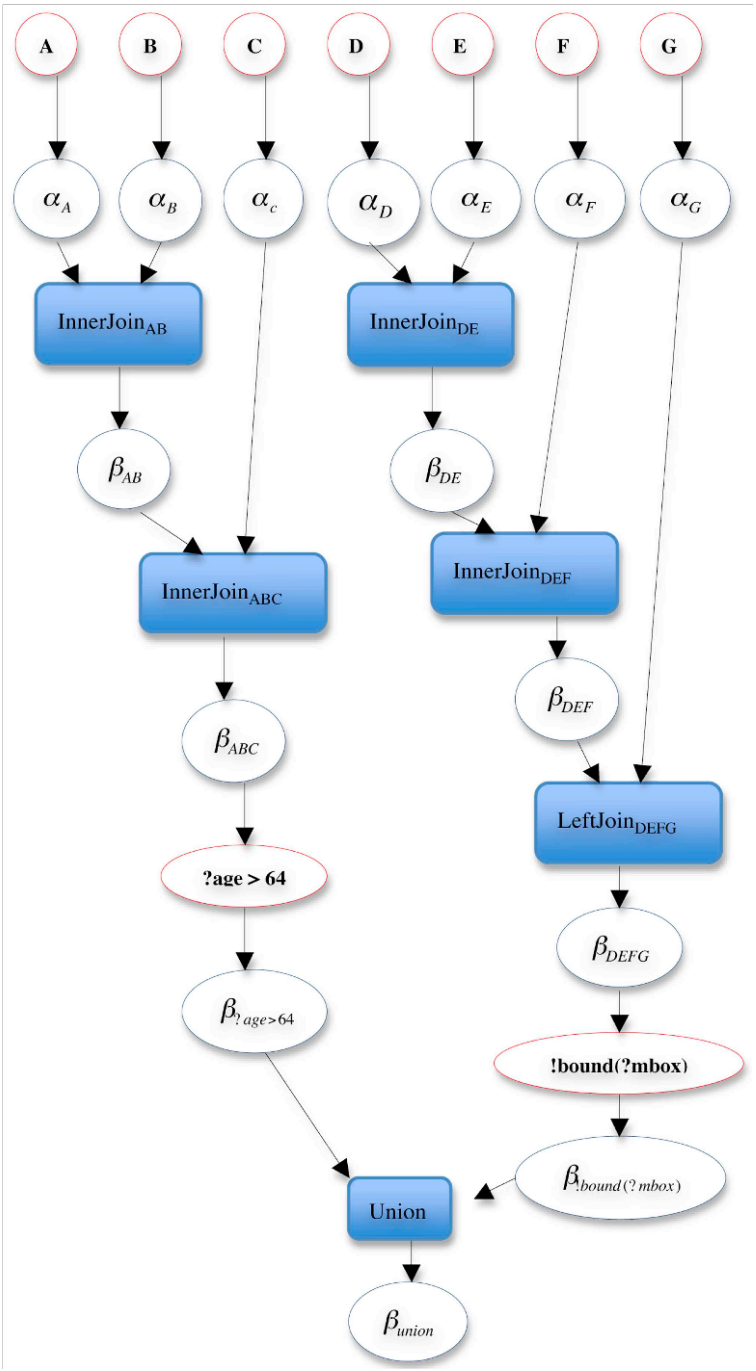

Figure 5b: Rete Network compiled from Figure 5a

Consider the case when a token resides in the memory node $\beta_{!bound(?mbox)}$. When a left token from β_{DEF} and a right token from α_G propagate to the two-input join node LeftJoinDEFG, these left and right tokens are joined together if their variable bindings are consistent. One of the variables bound to the RDF triples in the newly joined token is ?mbox. Before the joined token continues propagation to memory node β_{DEFG}, one-input filter node !bound(?mbox), and memory node $\beta_{!bound(?mbox)}$, all matching copies of the left token are removed from these successor nodes mentioned above in order to make way for the addition of this newly joined token. Assume for a moment that the token assumed to reside in $\beta_{!bound(?mbox)}$ matches the left token, then the token residing in $\beta_{!bound(?mbox)}$ is deleted. There is only one delete per memory node, no matter how many tokens match the left token in question. After all matching tokens of the left token are eradicated from subsequent memory nodes, the join node may now propagate and be added to any memory node it encounters. However, once the joined token, which contains a binding to variable ?mbox, propagates to the !bound(?mbox) node, the joined token is rejected and propagation halts because !bound(?mbox) means that no token that passes through should have a binding to this variable, ?mbox. Because propagation halts for this joined token, the matching copy of the left token that was deleted from $\beta_{!bound(?mbox)}$ will never be replaced with this newly joined token. This scenario is an explanation, by example, of our implementation of antijoin behavior as negation as failure in logic programming in our version of the Rete match algorithm.

While it is anticipated that SPARQL 1.1 will have an explicit negation operator, the fact that SPARQL 1.1 is still in development (at the time of this writing) encourages us to define Rete operators suitable for SPARQL 1.0 and not 1.1. Since we have based our definition of Diamonds LeftJoin (a left outer join operator for SPARQL OPTIONAL) through Gupta and Mumicks definition of outer join [19] and because Filter is defined as

Qian and Wiederholds relational select [32] whose constraint is !bound(?var), we may, by transitivity, consider the appropriate combination of these operators (as defined by [2]) a correct implementation of antijoin, which is used to provide incremental recomputation and consequently assists in the implementation of failure as negation in SPARQL. It is to note that the entire listing of our Rete operators and their formal definitions can be found in the appendix. Additionally we have an annotated query, seen in Figure 5(a), that utilizes all Rete match operators with the exception of *SolutionSequence* and *Intersection* and the Rete network for this annotated query is shown in Figure 5(b). The intersection operator is not shown in the network presented in Figure 5(b) because it is used to join two sub-constraints that are conjuncted together in a Filter expression. The query in Figure 5(a) contains no filter expressions that contain a conjunction. Please refer to the definition of Filter for further detail on how the intersection operator is used. The SolutionSequence takes the tokens from the last beta memory in a SPARQL query and projects the RDF triples in the tokens that are bound to variables in the select list in the SPARQL query. This operator is based on Qian and Wiederhold's relational project operator. Please refer to the appendix for further details.

Diamonds Rete operators, which implement the SPARQL language, are based on the definitions of Qian and Wiederhold [32] and Gupta and Mumick [19] and these definitions are implemented by a specific Java architecture (as seen by our UML class diagram in Figure 6), which allow our linked data engine to run. The following section will define Diamonds architecture in more detail.

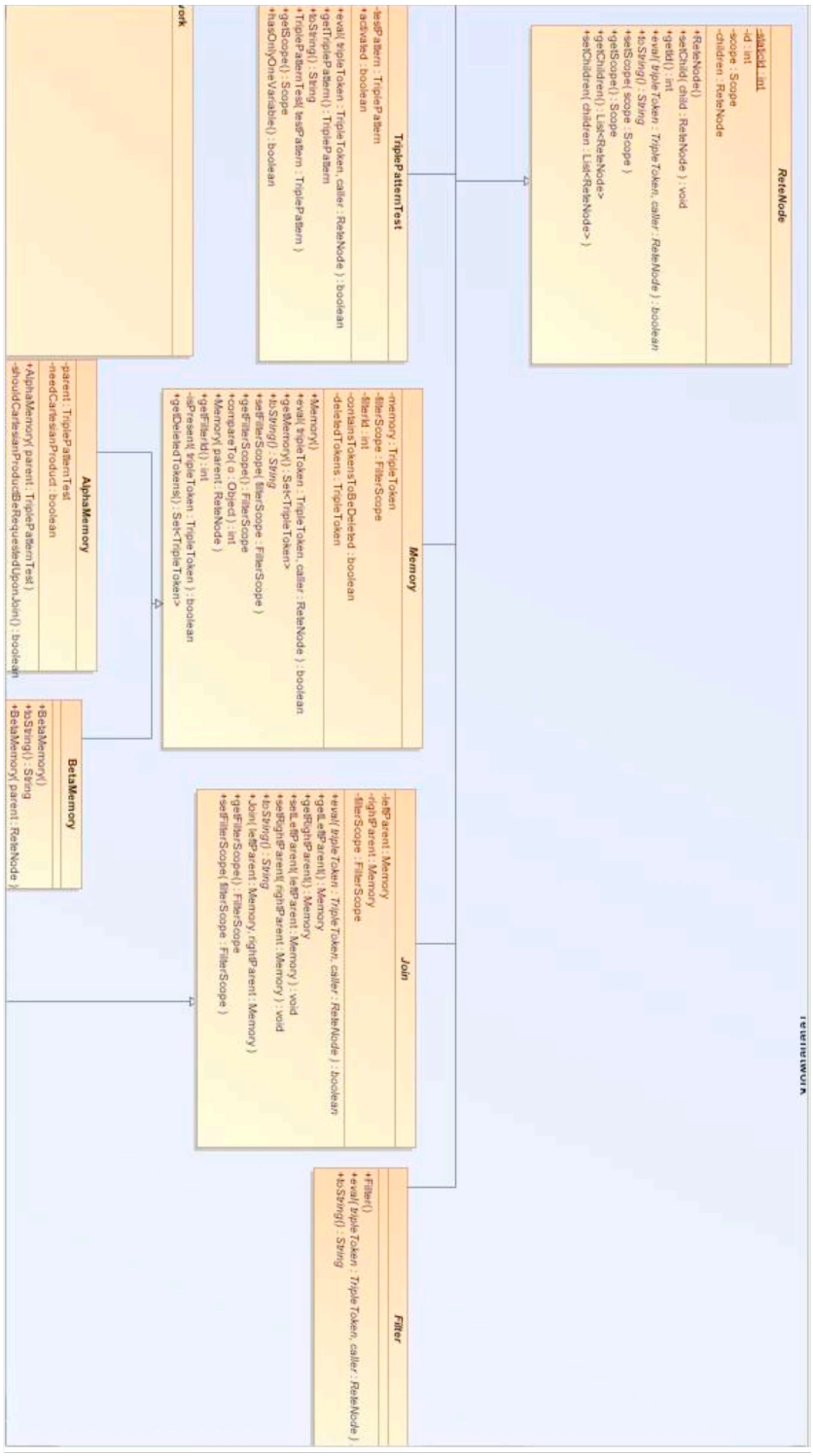

Figure 6: UML Java Class Diagram for Rete Operators

Chapter 5

Architecture and Implementation

In this section, we will explain the architecture of our system and how it caters to a continuous querying model. We also briefly describe the implementation details associated with Diamond's construction.

5.1 ARCHITECTURE

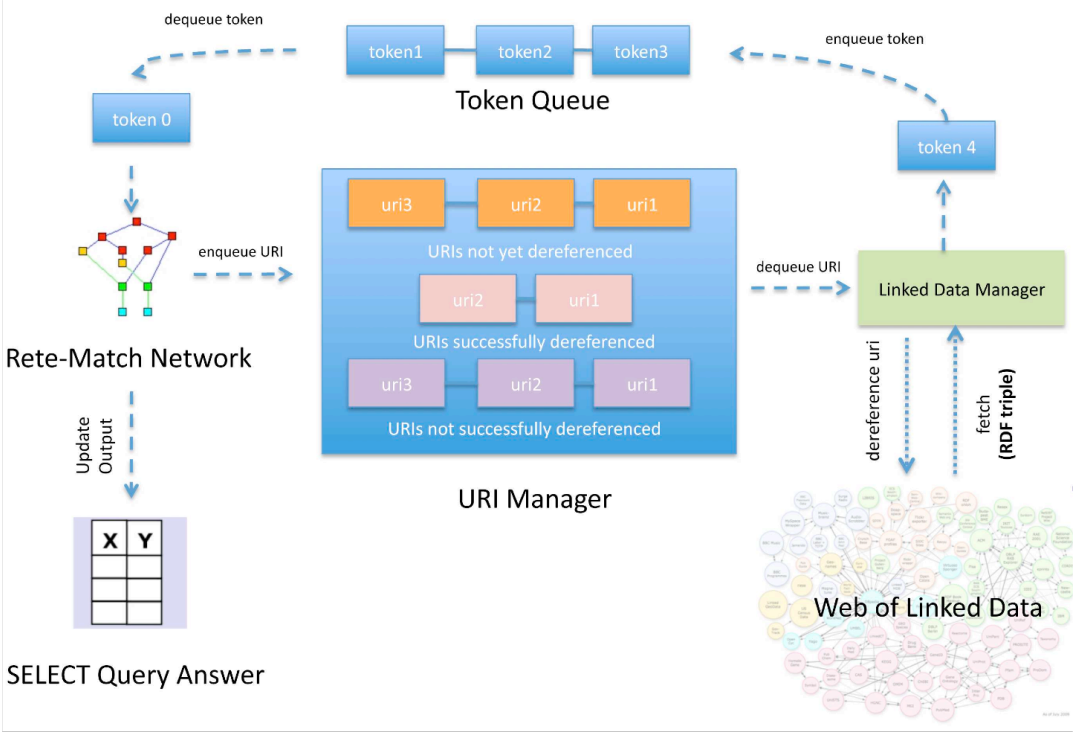

Figure 7: High Level Architectural Diagram of Diamond's Behavior

Figure 7 shows a high-level diagram of Diamond as it queries the web of linked data. Our system initially compiles a given SPARQL query, provided by the user, into an equivalent Rete-Match network shown on the far left of Figure 8. URIs from the query are initially populated in a queue of URIs that will be dereferenced later. There are two

other URI queues for future use; one queue that contains URIs unsuccessfully dereferenced, and the other queue for successfully dereferenced URIs. These URI queues can be found in a component called a *URI Manager*. Whenever a URI is enqueued to the URI Manager, the URI will always go to the correct queue. A *Linked Data Manager* will dequeue URIs and use it to dereference RDF triples from a server for which the URI points. RDF triples are converted, by the Linked Data Manager, into *triple tokens* or tokens for short. Tokens are tuples that contain a list of RDF triples bound by their triples patterns and a +/- tag that signals the token to be added to the state of the match network or to assist in token removal from the network. Tokens are stored in a queue called the *Token Queue* and dequeue to propagate through the Rete match network. As tokens propagate through the network, join nodes have the opportunity to integrate these tokens together depending on variable consistency. The tokens that make it to the bottom most memory will be used to formulate an answer set in the form of a table created by a SELECT query. If tokens are partially matched to the match network (which means an RDF triple is matched to a SPARQL graph pattern), then URIs contained within the RDF triple(s) of the token are enqueued to the URI queue to be dereferenced later. Fetching, evaluation, and incremental updates are a cyclic process and are treated as such in the system.

5.2 IMPLEMENTATION

The operators, algorithms and data structures described in previous sections are proven in practice through the construction and successful execution of a Java implementation of the linked data query engine. We choose Java because of the language's well-documented portability on multiple computing environments. When a user submits a SPARQL query to the system a BNF grammar description of the SPARQL

language is used by JavaCC[4] to construct a lexer and top-down parser[5]. JavaCC is limited to LL(k) grammars and SPARQL is well-suited for such a tool as it is considered an LL(1) grammar [31]. Diamond uses Java Tree Builder[6] (JTB) to construct a syntax tree in conjunction with JavaCC. JTB allows us to traverse a SPARQL syntax tree using the visitor pattern through JTB generated visitor classes. While visiting each syntax tree node, Diamond incrementally constructs an expression tree, where each node represents a SPARQL graph pattern or filter expression. The nodes point to any sub-patterns in the query. Finally Diamond constructs the Rete match network from this expression tree and useful URIs gotten from the triple patterns in the query (if any) are enqueued into the URI Manager. Diamond uses an open source semantic web framework called Sesame[7] to dereference URIs and extract RDF triples (if any). Sesame is useful for executing HTTP requests and RDF data parsing in a variety of RDF syntax (e.g. NTriples, Notation3, Turtle, etc.). Our network constructed from our Rete match operator set processes the tokens acquired. Each operator is a node in the network and is represented in an inheritance class structure where `ReteNode` is an abstract parent class, while other nodes extend `ReteNode` such as `InnerJoin`, `LeftJoin`, `Union`, `Intersection`, and `Filter` as seen in Figure 7. While Diamond is a fairly complex system, it is well documented with accompanying Java documentation and UML class diagrams.

[4] Third-Party Open Source software publicly available at `https://javacc.dev.java.net`

[5] Other SPARQL processing systems, most notably Jena, includes SPARQL parsers, but they are intimately tied to their respective systems and of little use to scientists/developers looking for stand-alone functioning parsers for use out of the box. Diamonds SPARQL parser is a wonderful contribution to the semantic web community due to its independence and out of the box execution.

[6] Third-Party open source software available at http://compilers.cs.ucla.edu/jtb

[7] Publicly available at http://www.openrdf.org

Chapter 6

Evaluation

Historically, Rete match networks can be treated as physical query plans that can be optimized [28]. We hypothesize that our Rete match networks will also be susceptible to optimizations and these optimizations will have a positive impact on SPARQL query execution. To prove this claim, we executed a set of experiments using simple optimizations that show a speed increase in SPARQL query executions. Additionally, because our Rete match networks are based on relational operators that can incrementally recomputed data, we also prove that our Rete match networks are equivalent in expressive power to the core of SPARQL.

6.1 Optimizations

Optimizations of physical query plans in relational databases using a variety of cost estimation and heuristics to decide join operator placement is not new [18]. We try one heuristic to speed up SPARQL query processing by optimizing our Rete match network. Whenever a filter node appears towards the bottom of the network, we look for ways to raise the filter node closer to the top of the network without changing the semantics of the query. This is akin to ”pushing down selects” in a relational tree, except in our case the tree is upside down and the Filter operator acts as our ”select”. Filter location is important as filter nodes towards the top of the network are able to eliminate tokens early before avoiding unnecessary propagation through the rest of the network and unnecessary incremental recomputation.

For our experiment, we have generated BSBM data sets of different sizes (a varying amount of triples) using the BSBM data generator [9] using several modified

BSBM queries on each data set. Each query has an optimized version (where filter nodes are towards the top of the network) and an un-optimized version (where filter nodes are towards the bottom of the network). We tested both query versions on each data set. We execute every experiment ten times (excluding three warm-up runs) and calculate the average query execution speed in seconds. According to our analysis, seen in Figure 8, our approach allows an increase in query performance 86% of the time, proving that SPARQL queries are positively affected by network optimizations through operator re-arrangement.

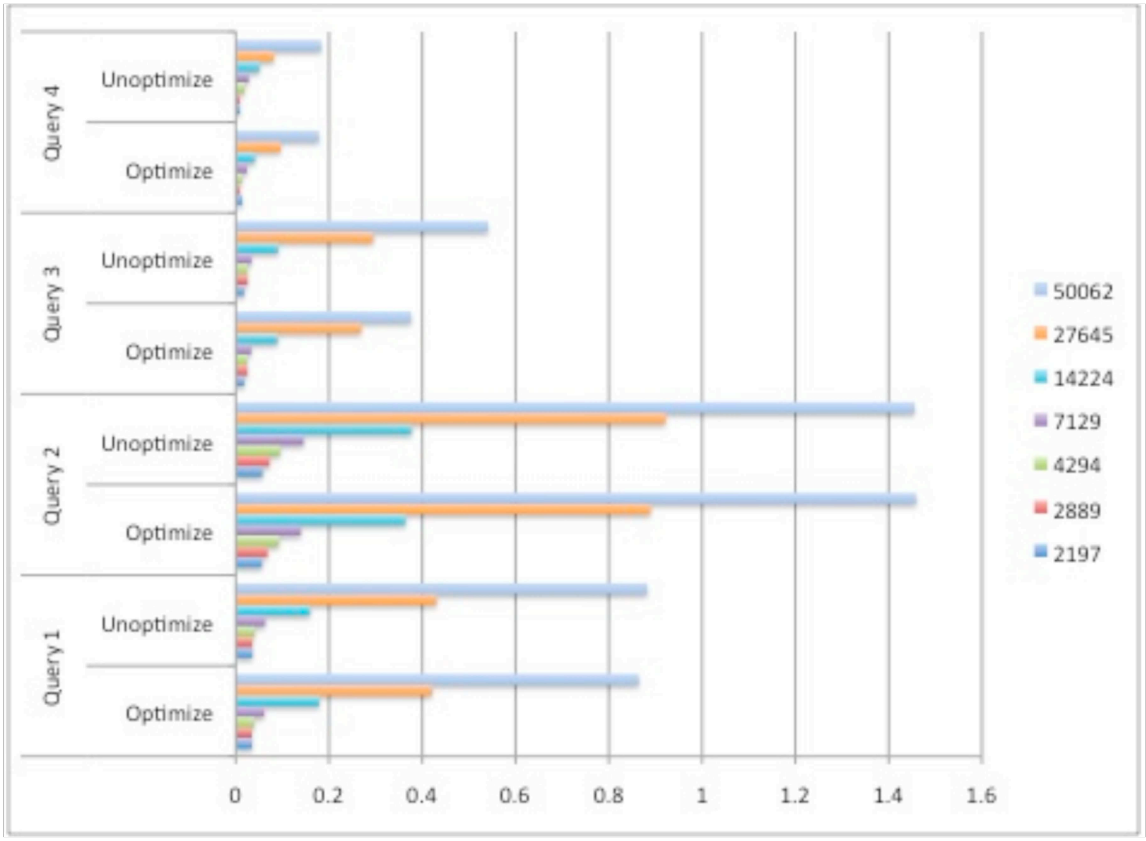

Figure 8: Results in Seconds for Un-optimized and Optimized Queries varied by number of triples

Experiments were performed locally on a machine with two Intel Core 2 E6300 processors at 1.86GHz and 2GB of main memory on Ubuntu Linux. While optimization experiments are executed locally, we also assess the general performance of Diamond in the wild by running linked data experiments on three queries (as seen in Table 2).

Query	Query 1	Query 2	Query 3
# Of Results	10	1	4
# of Tokens Processed	10,246	10,106	64
Execution Time	5 min, 47 sec	3 min, 43 sec	2.12 sec

Table 2: Timing Results from Linked Data Queries

Query 1: Determine the interests of people whom Tim Berners-Lee knows[8]

Query 2: Determine the name, latitude and longitude of the birthplace of the 43rd President of the U.S., George W. Bush.

Query 3: Determine the location of a specific church in Paris, France.[9]

One interesting heuristic we adopt from [21] is the idea that URIs contained within an RDF triple that partially matches a query should be dereferenced as soon as possible, giving that URI higher priority over others seen. In the case of Diamond, a token that passes at least one test operator in our Rete match is akin to partially satisfying a SPARQL query. Therefore, we make any URIs contained within that token of high priority within our URI queue in the URI Manager mentioned above. Thus, these URIs will be given preference to dereference first.

Finally, it is to note that in order to gain some confidence in our results, we tested Diamond using the RAP test suite[10] and passed 126 of 151 test cases. The other 25 test

[8] Query acquired from [21] for experimentation purposes.

[9] Query modified and borrowed from http://docs.openlinksw.com/virtuoso/rdfsparqlgeospat.html. The results of the query reveal churches of this name in four different languages, hence four results are produced.

cases ran ASK queries, DESCRIBE queries, CONSTRUCT queries and solution modifiers, which Diamond does not support because it is not core to SPARQL and not part of our study.

6.2 EXPRESSIVE EQUIVALENCE

Formal Rete match operator definitions can be found in the appendix. SPARQL is equivalent to relational algebra [2]. We have defined our operator set based on relational operators and proved their correctness through the propagation rules expressed in the Appendix for each operator set. Since SPARQL is equivalent in expressive power to relational algebra, and our operator set is defined on that algebra, and since our Rete networks are defined by the operator set, we may conclude, by transitivity, that our Rete match networks are equivalent in expressive power to SPARQL.

[10] Test suite located at http://www.seasr.org/wp-content/plugins/meandre/rdfapi-php/doc/

Chapter 7

Future Work and Conclusions

Hopefully our optimization studies have convinced the reader that using our system is worthwhile in order to try out their own optimization strategies. Also, we look forward in future iterations of this work to try out parallel Rete match algorithms, and lazy match algorithms. Finally, we would like to extend our study of SELECT queries by implementing ASK, DESCRIBE, solution modifiers, etc.

This paper presents a match system defined by an operator set. We show that our networks can be optimized and provide evidence showing that optimizing match networks effectively improves SPARQL query performance. We also provide a well-engineered linked data query engine written in Java. Finally, we show that our networks are equivalent in expressive power to the core of SPARQL.

Appendix

Triple tokens contain bound RDF triples. For the purposes of our proof we'll consider, without loss of generality, RDF triples as data to be consumed by our operator set. Assume *tp* is a triple pattern, *t* is an RDF triple and *V* is an infinite set of variables as defined in [30, 31]. Assume *S*, *R* and *T* are sets of RDF triples. Two partial operators we use to supplement our discussion, from [32], are

$$\textit{Disjoint union } R \oplus S = \begin{cases} R \cup S, \text{if } R \text{ and } S; \\ \text{not defined, otherwise.} \end{cases}$$

$$\textit{Contained difference } R \ominus S = \begin{cases} R - S, \text{if } R \subseteq S; \\ \text{not defined, otherwise.} \end{cases}$$

NOTATIONS

The notations below are used as accessors to portions of triple patterns and RDF triple atoms (i.e. – subject, predicate, and object).

Notation 1 *tp.s* stands for the subject of triple pattern *tp*

Notation 2 *t.s* stands for the subject of triple pattern *tp*

Notation 3 *tp.p* stands for the predicate of triple pattern *tp*

Notation 4 *t.p* stands for the predicate of RDF triple *t*

Notation 5 *tp.o* stands for the object of triple pattern *tp*

Notation 6 *t.o* stands for the object of RDF triple *t*

RETE NETWORK OPERATOR DEFINITIONS AND THEOREMS

Definition 1

$$\sigma_{tp}(R) = \begin{cases} R, \ if \ tp.s \in V \wedge tp.p \in V \wedge tp.o \in V \ ; \\ \{t : t \in R \wedge tp.o = t.o\}, \ if \ tp.s \in V \wedge tp.p \in V \wedge tp.o \notin V \ ; \\ \{t : t \in R \wedge tp.p = t.p\}, \ if \ tp.s \in V \wedge tp.p \notin V \wedge tp.o \in V \ ; \\ \{t : t \in R \wedge tp.p = t.p \wedge tp.o = t.o\}, \ if \ tp.s \in V \wedge tp.p \notin V \wedge tp.o \notin V \ ; \\ \{t : t \in R \wedge tp.s = t.s\}, \ if \ tp.s \notin V \wedge tp.p \in V \wedge tp.o \in V \ ; \\ \{t : t \in R \wedge tp.s = t.s \wedge tp.o = t.o\}, \ if \ tp.s \notin V \wedge tp.p \in V \wedge tp.o \notin V \ ; \\ \{t : t \in R \wedge tp.s = t.s \wedge tp.p = t.p\}, \ if \ tp.s \notin V \wedge tp.p \notin V \wedge tp.o \in V \ ; \\ \{t : t \in R \wedge tp.s = t.s \wedge tp.p = t.p \wedge tp.o = t.o\}, otherwise. \end{cases}$$

Definition 1 selects all RDF triples in set *R*, if the triple pattern contains all variables, otherwise all RDF triples in set R that match literals/URIs of triple pattern *tp* are selected.

Definition 2

$$\sigma_{tp}^{\mathcal{D}}(R) = \sigma_{tp}(R)$$

The Rete operator, *InnerJoin*, is based on a relational join operator that joins two sets of RDF triples, *R* and *S*, together.

Definition 3

$$R \bowtie_{\mathcal{D}} S = R \bowtie S$$

The Rete operator, *InnerJoin*, is based on a relational join operator that joins two sets of RDF triples, *R* and *S*, together.

Definition 4

$$R \ltimes_{\mathcal{D}} S = R \ltimes S$$

The Rete operator, *LeftJoin*, is based on a relational left outer join operator that joins two sets of RDF triples, *R* and *S*, together.

Definition 5

$$R \cup_{\mathcal{D}} S = R \cup S$$

The Rete operator, *Union*, is based on a set union operator that unions two sets of RDF triples, *R* and *S*, together.

Definition 6

$$R \cap_{\mathcal{D}} S = R \cap S$$

The Rete operator, *Intersect*, is based on a set intersection operator that intersects two sets of RDF triples, *R* and *S*, together.

Definition 7

$$\pi_s^{\mathcal{D}}(R) = \pi_s(R)$$

The Rete operator, *SolutionSequence*, is based on a relational project that extracts the bindings of a list of variables, *s*, from a set of RDF triples, *R*.

Definition 8

$$\mathbf{F}_c(\mathcal{R}) = \begin{cases} \mathbf{F}_x(\mathcal{R}) \ \cap_D \ \mathbf{F}_y(\mathcal{R}), if \ c \ has \ form \ x \&\& \ y; \\ \mathbf{F}_x(\mathcal{R}) \ \cup_D \ \mathbf{F}_y(\mathcal{R}), if \ c has \ form \ x || \ y; \\ \sigma_c(\mathcal{R}), otherwise. \end{cases}$$

The Rete operator, *Filter*, is a recursively defined operator based on relational algebraic selection of a set of RDF triples, *R*, using a constant *c*. *x* and *y* are sub-constraints of *c*.

Theorem 1

$$(((F_c(R) = R_1) \wedge (\sigma_c(R) = R_2)) \Rightarrow (R_1 = R_2)) \vee$$
$$(((F_{x\&\&y}(R) = R_1) \wedge (F_x \cap_D F_y(R) = R_2)) \Rightarrow (R_1 = R_2))) \vee$$
$$(((F_{x||y}(R) = R_1) \wedge (F_x \cup_D F_y(R) = R_2) \Rightarrow (R_1 = R_2))$$

where R, R_1, R_2 are relations.

Proof

Statement P(n): We prove that Theorem 7 holds by induction over *n* basic constraints where $n \in N^*$ and $N^* = \{1,2,3,...\}$.

Base Case P(1):

1. Assume only one basic constraint, *c*.
2. Assume $F_c(R) = R_1$
3. Assume $\sigma_c(R) = R_2$
4. $F_c(R) = R_1 \wedge \sigma_c(R) = R_2$ (By lines 2 and 3)
5. Since *c* is the only basic constraint, then it does not have the form *x && y* nor the form *x // y*, where *x* and *y* are sub-constraints of *c*. (By definition of basic constraint and line 1)
6. $F_c(R) = \sigma_c(R)$ (By definition 11 and line 5)
7. $R_1 = \sigma_c(R)$ (By lines 6 and 2)
8. $R_1 = R_2$ (By lines 7 and 3)
9. P(1) holds (by lines 4 and 8)

Inductive Case P(n+1):

10. Assume P(n) holds.
11. Sub-case 1:
    a. Consider the case when $F_{c\&\&z}(R) = R_1$ where *c&&z* is a constraint and *z* is a basic constraint.

b. So c && z in $F_{c\&\&z}(R)$ contains n+1 basic constraints, where c has n basic constraints and z has one basic constraint. (By line 11a).

c. Then $F_{c\&\&z}(R) = F_c(R) \cap_D F_z(R)$. (By definition 11 and line 11b)

d. $F_c(R)$ evaluates to some relation R' (by induction hypothesis)

e. $F_z(R) = \sigma_z(R)$ which evaluates to some relation R''. (By definition 11 and definition of relational algebra operator select)

f. $F_{c\&\&z}(R) = R' \cap_D F_z(R)$ (By lines 11d and 11c)

g. $F_{c\&\&z}(R) = R' \cap_D R''$ (By lines 11e and 11f)

h. $R' \cap_D R'' = R' \cap R''$ (By definition 10)

i. $R' \cap R''$ evaluates to some relation R_2. (By definition of set union)

j. $F_{c\&\&z}(R) = R_2$ (By lines 11g, 11h, 11i)

k. $R_1 = R_2$ (By lines 11a and 11h)

12. Sub-case 2

a. Consider the case when $F_{c||z}(R) = R_1$ where c // z is a constraint and z is a basic constraint.

b. We can now prove that if $F_{c||z}(R) = R_1$ and $F_c(R) \cup_D F_z(R) = R_2$, then $R_1 = R_2$ using a similar proof as the one expressed in Sub-case 1.

13. Thus, Theorem 4.7 holds.

Q.E.D.

PROPAGATION RULES

The following is a set of equivalence preserving transformation rules that allow each Rete operator to perform incremental recomputation every time a single RDF triple enters the Rete match system (in the form of a triple token). We build these rules in the form of [46] because our Rete operators are based on the same relational operators used in [46] to perform incremental recomputation.

TriplePatternTest Rules

1. $\sigma^{\mathcal{D}}_{tp}(R \ominus S) \Longrightarrow \sigma^{\mathcal{D}}_{tp}(R) \ominus \sigma^{\mathcal{D}}_{tp}(S)$
2. $\sigma^{\mathcal{D}}_{tp}(R \oplus S) \Longrightarrow \sigma^{\mathcal{D}}_{tp}(R) \oplus \sigma^{\mathcal{D}}_{tp}(S)$

SolutionSequence Rules

1. $\pi^{\mathcal{D}}_{s}(R \ominus S) \Longrightarrow \pi^{\mathcal{D}}_{s}(R) \ominus \pi^{\mathcal{D}}_{s}(S)$
2. $\pi^{\mathcal{D}}_{s}(R \oplus S) \Longrightarrow \pi^{\mathcal{D}}_{s}(R) \oplus \pi^{\mathcal{D}}_{s}(S)$

Union Rules

1. $(R \ominus S) \cup_{\mathcal{D}} T \Longrightarrow (R \cup_{\mathcal{D}} T) \ominus (S - T)$
2. $(R \oplus S) \cup_{\mathcal{D}} T \Longrightarrow (R \cup_{\mathcal{D}} T) \oplus (S - T)$

Intersection Rules

1. $(R \ominus S) \cap_{\mathcal{D}} T \Longrightarrow (R \cap_{\mathcal{D}} T) \ominus (S \cap_{\mathcal{D}} T)$
2. $(R \oplus S) \cap_{\mathcal{D}} T \Longrightarrow (R \cap_{\mathcal{D}} T) \oplus (S \cap_{\mathcal{D}} T)$

InnerJoin Rules

1. $(R \ominus S) \bowtie_{\mathcal{D}} T \Longrightarrow (R \bowtie_{\mathcal{D}} T) \ominus (S \bowtie_{\mathcal{D}} T)$
2. $(R \oplus S) \bowtie_{\mathcal{D}} T \Longrightarrow (R \bowtie_{\mathcal{D}} T) \oplus (S \bowtie_{\mathcal{D}} T)$

LeftJoin Rules

1. $(R \ominus S) ⟕_{\mathcal{D}} T \Longrightarrow (R ⟕_{\mathcal{D}} T) \ominus (S ⟕_{\mathcal{D}} T)$
2. $(R \oplus S) ⟕_{\mathcal{D}} T \Longrightarrow (R ⟕_{\mathcal{D}} T) \oplus (S ⟕_{\mathcal{D}} T)$

AntiJoin Rules

1. $\mathbf{F}_c((R \ominus S) ▷ T) \Longrightarrow \mathbf{F}_c(R ▷ T) \ominus \mathbf{F}_c(S ▷ T)$
2. $\mathbf{F}_c((R \oplus S) ▷ T) \Longrightarrow \mathbf{F}_c(R ▷ T) \oplus \mathbf{F}_c(S ▷ T)$

where c = !bound(?x) and ?x is a variable that can only be bound to RDF triple objects in T (if at all)

Theorem 2

The propagation rules are equivalence preserving.

Proof.

Theorem 1 of [46] proves the propagation rules of select, project, union intersect and join are equivalence preserving. Since our Rete operators such as *TriplePatternTest, SoltuionSequence, Union, Intersection, InnnerJoin,* and *Filter* perform incremental evaluation using the Rete-Match algorithm and are based on the same relational algebra operators mentioned previously, as seen in the definitions above, then a similar proof may be derived as in theorem 1 of [46] for the propagation rules of our Rete operators; *TriplePatternTest*, *SolutionSequence*, *Union*, *Intersection*, *InnerJoin*, and *Filter*. Consequently, these rules are equivalence preserving.

Additionally, since the Rete-Match saves state changes between processing iterations similar to change tables in incremental maintenance operations of outerjoin as seen in [47] then our *LeftJoin* propagation rule, which uses the Rete operator *LeftJoin* based on relational left outer join, is also equivalence preserving.

Finally, SPARQL 1.0 does not contain an anti join operator in its SPARQL algebra. However in SPARQL 1.0, anti join is evaluated as failure in logic programming using pattern (P_1 OPT P_2) FILTER (!bound(?x)) where ?x is only expressed in pattern P_2 according to [2]. This pattern can be equivalently expressed using the Rete-Match algebraic expression $\mathbf{F}_c(R \bowtie_D S)$ where *R*, *S* are sets of RDF triples (to be evaluated by P_1 and P_2 respectively in our previous SPARQL pattern_ and *c* is the constraint !bound(?x) such that ?x can only be bound to RDF triples in *S* (if at all). Since we have proven that $\mathbf{F}_c$ and $\bowtie_D$ are both Rete operators with propagation rules that are equivalence preserving above, by transitivity, the anti join propagation rules are also equivalence preserving.

Therefore, from all of the above, our propagation rules are equivalence preserving.

Q.E.D.

Theorem 3

Our Rete-Match networks are equivalent in expressive power to SPARQL.

Proof.

SPARQL is equivalent to relational algebra [2]. We have defined our Rete operator set based on relational algebra operators and proved their correctness through a series of propagation rules matching each Rete operator. Our rete-Match networks are constructed using the Rete operators defined within this paper. Therefore, by transitivity, since SPARQL is equivalent in expressive power to relational algebra and since relational algebra operators define our Rete network operators and since those network operators compose a Rete-Match network, we may conclude that Rete-Match networks are equivalent in expressive power to SPARQL.

Q.E.D.

References

[1] Daniel J. Abadi, Don Carney, Ugur Cetintemel, Mitch Cherniack, Christian Convey, Sang- don Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik, Aurora: a new model and architecture for data stream management, The VLDB Journal 12 (2003), no. 2, 120–139.

[2] Renzo Angles and Claudio Gutierrez, The expressive power of sparql, ISWC '08: Proceedings of the 7th International Conference on The Semantic Web (Berlin, Heidelberg), Springer- Verlag, 2008, pp. 114–129.

[3] Mostafa M. Aref and Mohammed A. Tayyib, Lana—match algorithm: a parallel version of the rete—match algorithm, Parallel Comput. 24 (1998), no. 5-6, 763–775.

[4] Davide F. Barbieri and Emanuele Della Valle, A proposal for publishing data streams as linked data - a position paper, Linked Data on the Web (Raleigh, North Carolina), April 2010.

[5] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus, C-sparql: Sparql for continuous querying, WWW '09: Proceedings of the 18th international conference on World wide web (New York, NY, USA), ACM, 2009, pp. 1061– 1062.

[6] Don Batory, The leaps algorithm, Tech. Report CS-TR-94-28, Austin, TX, USA, 1994.

[7]Tim Berners-Lee, Design issues: Linked data, July 2006. `http://www.w3.org/DesignIssues/LinkedData.html`, (retrieved September 9, 2010)

[8] Tim Berners-Lee, Yuhsin Chen, Lydia Chilton, Dan Connolly, Ruth Dhanaraj, James Hollen- bach, Adam Lerer, and David Sheets, Tabulator: Exploring and analyzing linked data on the semantic web, The 3rd International Semantic Web User Interaction Workshop (2006).

[9] Christian Bizer, Christian Bizer, and Andreas" Schultz, The berlin sparql benchmark, International Journal On Semantic Web And Information Systems (2009).

[10] Christian Bizer, Tom Heath, and Tim Berners-Lee, Linked data - the story so far, Journal on Semantic Web and Information Systems 5 (2009), no. 3, 1–22.

[11] Andre Bolles, Marco Grawunder, and Jonas Jacobi, Streaming sparql extending sparql to pro- cess data streams, ESWC'08: Proceedings of the 5th European semantic web conference on The semantic web (Berlin, Heidelberg), Springer-Verlag, 2008, pp. 448–462.

[12] Gianluca Correndo, Manuel Salvadores, Ian Millard, Hugh Glaser, and Nigel Shadbolt, Sparql query rewriting for implementing data integration over linked data, Proceedings of the 2010 EDBT/ICDT Workshops (New York, NY, USA), EDBT '10, ACM, 2010, pp. 4:1–4:11.

[13] Richard Cygniak, A relational algebra for sparql, Tech. Report HPL-2005-170, HP-Labs, 2005. [14] Samur F. C. de Araujo and Daniel Schwabe, Explorator: a tool for exploring rdf data through direct manipulation, Linked Data on the Web (Madrid, Spain), April 2009.

[15] Samur F. C. de Araujo, Daniel Schwabe, and Simone D. J. Barbosa, Experimenting with explorator: a direct manipulation generic rdf browser and querying tool, Visual Interfaces to the Social and The Semantic Web (Sanibel Island, Florida, USA), February 2009.

[16] Charles Lanny Forgy, On the efficient implementation of production systems., Ph.D. thesis, Pittsburgh, PA, USA, 1979.

[17] Charles Lanny Forgy, Rete: A fast algorithm for the many pattern/many object pattern match problem, Journal on Artificial Intelligence 19 (1982), no. 1, 17–37.

[18] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom, Database systems: The complete book, 2 ed., Prentice Hall Press, Upper Saddle River, NJ, USA, 2008.

[19] Himanshu Gupta and Inderpal Singh Mumick, Incremental maintenance of aggregate and outerjoin expressions, Inf. Syst. 31 (2006), 435–464.

[20] Eric Hanson, Nabeel Al-fayoumi, Chris Carnes, Mohktar Kandill, Huisheng Liu, Min Lu, J.B. Park, and Albert Vernon, Triggerman: An asynchronous trigger processor as an extension to an object-relational dbms, Tech. Report TR 97-024, University of Florida, 1997.

[21] Olaf Hartig, Christian Bizer, and Johann-Christoph Freytag, Executing sparql queries over the web of linked data, ISWC '09: Proceedings of the 8th International Semantic Web Conference (Berlin, Heidelberg), Springer-Verlag, 2009, pp. 293–309.

[22] Olaf Hartig and Ralf Heese, The sparql query graph model for query optimization, Proceedings of the 4th European conference on The Semantic Web: Research and Applications (Berlin, Heidelberg), ESWC '07, Springer-Verlag, 2007, pp. 564–578.

[23] Olaf Hartig and Andreas Langegger, A database perspective on consuming linked data on the web, Datenbank-Spektrum (2010), 1–10, 10.1007/s13222-010-0021-7.

[24] Chun Jin and Jaime Carbonell, Argus: Efficient scalable continuous query optimization for large-volume data streams, IDEAS '06: Proceedings of the 10th International Database Engineering and Applications Symposium (Washington, DC, USA), IEEE Computer Society, 2006, pp. 256–262.

[25] Graham Klyne, Jeremy J. Carroll, and Brian McBride, Resource description framework (rdf): Concepts and abstract syntax. February 2004, `http://www.w3.org/TR/rdf-concepts/`, (retrieved August 24, 2010)

[26] Joseph B. Kopena, Joseph B. Kopena, and William C." Regli, Damljesskb: A tool for reasoning with the semantic web, IEEE INTELLIGENT SYSTEMS 18 (2003), 74–77.

[27] Milind Mahajan and V. K. Prasanna Kumar, Efficient parallel implementation of rete pattern matching, Comput. Syst. Sci. Eng. 5 (1990), no. 3, 187–192.

[28] Daniel P. Miranker, David A. Brant, Bernie Lofaso, and David Gadbois, On the performance of lazy matching in production systems, Proceedings of the eighth National conference on Artificial intelligence - Volume 1, AAAI'90, AAAI Press, 1990, pp. 685–692.

[29] Daniel P. Miranker, Treat: a new and efficient match algorithm for ai production systems, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.

[30] Jorge P ´erez, Marcelo Arenas, and Claudio Gutierrez, Semantics and complexity of sparql, ACM Trans. Database Syst. 34 (2009), no. 3, 1–45.

[31] Eric Prud'hommeaux and Andy Seaborne, Sparql query language for rdf, January 2008, `http://www.w3.org/TR/rdf-sparql-query/`, (retrieved August 24, 2010)

[32] X. Qian and Gio Wiederhold, Incremental recomputation of active relational expressions, IEEE Trans. on Knowl. and Data Eng. 3 (1991), 337–341.

[33] Bastian Quilitz and Ulf Leser, Querying distributed rdf data sources with sparql, ESWC'08: Proceedings of the 5th European semantic web conference on The semantic web (Berlin, Heidelberg), Springer-Verlag, 2008, pp. 524–538.

[34] Michael Schmidt, Michael Meier, and Georg Lausen, Foundations of sparql query optimization, ICDT '10: Proceedings of the 13th International Conference on Database Theory (New York, NY, USA), ACM, 2010, pp. 4–33.

[35] Mohammed A. Tayyib, Parallelization of rete-match algorithm for distributed memory archi- tectures, Master's thesis, King Fahd University Of Petroleum And Minerals, July 1996.

[36] E. N. Hanson, An initial report on the design of ariel dbms with an integrated production rule system, SIGMOD Rec. 18 (1989), 12–19.

Vita

Rodolfo Kaplan Depena received his B.S. in Computer Sciences with a 2nd degree in Mathematics and a Business Foundations Certification (graduating with distinction), all from The University of Texas at Austin in 2008. During the summers he completed internships at ExxonMobil and JP Morgan Chase & Co. While going to school and performing scientific research, he also worked as a Resident Assistant counseling and mentoring undergraduate students in Moore-Hill Residence Hall and tutored students in the Department of Computer Sciences at UT-Austin. He took time to volunteer with the UTCS Roadshow and traveled to visit grade-school students in Texas promoting the Department of Computer Science and UT-Austin at large. In 2008 he re-entered The University of Texas at Austin in the Computer Science Masters program.

Permanent address: 12633 Memorial Drive #152, Houston, Texas 77024

This thesis was typed by Rodolfo Kaplan Depena.